Operating Systems

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/

File Systems

- File systems provide long-term information storage
- Must store large amounts of data
- Information stored must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently
- Two facets:
  - File system interface (logical view)
  - File system implementation (physical view)

The File Abstraction

- Files are named entities used by processes to store data
- Defined by a number of attributes
  - Name: File name, Extension
  - Type: Data, Directory, Special (Devices, IPC, etc)
  - Location: Info to map file to disk blocks
  - Size
  - Protection: Read, Write, Execute
  - Time(s): Creation, Modification, Last use
  - Owner: User ID, Group ID etc
- Usually seen as a stream of bytes (record-based structures are possible)

File Access

- Sequential access
  - Read all bytes/records from the beginning
  - Cannot jump around, could rewind or back up
  - Convenient when medium was tape
- Random access
  - Bytes/records read in any order
  - Essential for data base systems
  - Read can be …
    - move file marker (seek), then read or …
    - read and then move file marker
- Indexed access

File Operations

- Create
- Delete
- Open
- Close
- Read
- Write
- Append
- Seek
- Get attributes
- Set Attributes
- Rename

An Example Program


```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];

      if (argc != 3) exit(1);           /* syntax error if argc is not 3 */
```

An Example Program


```
    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

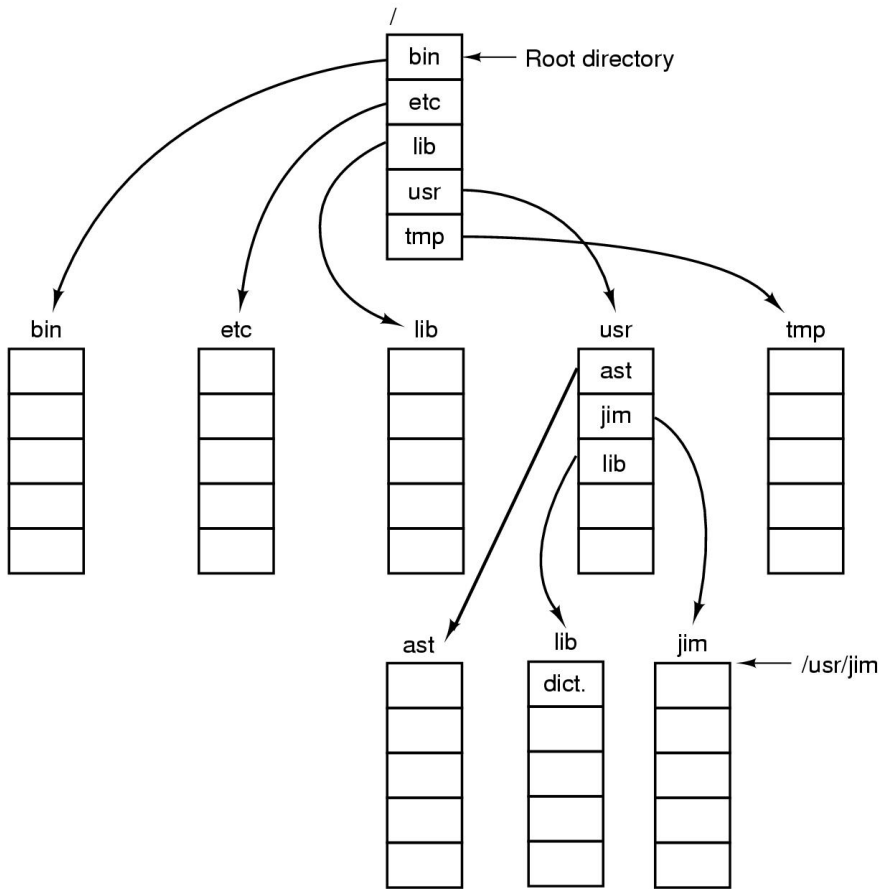
Memory-Mapped Files

- Some systems allow for mapping files into virtual memory
- File is then accesses as part of the memory
- If shared-memory is supported memory-mapped files can be accessed by multiple processes
- Solution can simplify access

Directories

- Used to organize files
- Usually organized in trees or directed graphs
- File is identified by a *path name* in the graph
  - absolute path name
  - relative path name (wrt working directory as stored in the PCB)
- "." and ".." entries are used to identify current and parent directories

A UNIX Directory Tree



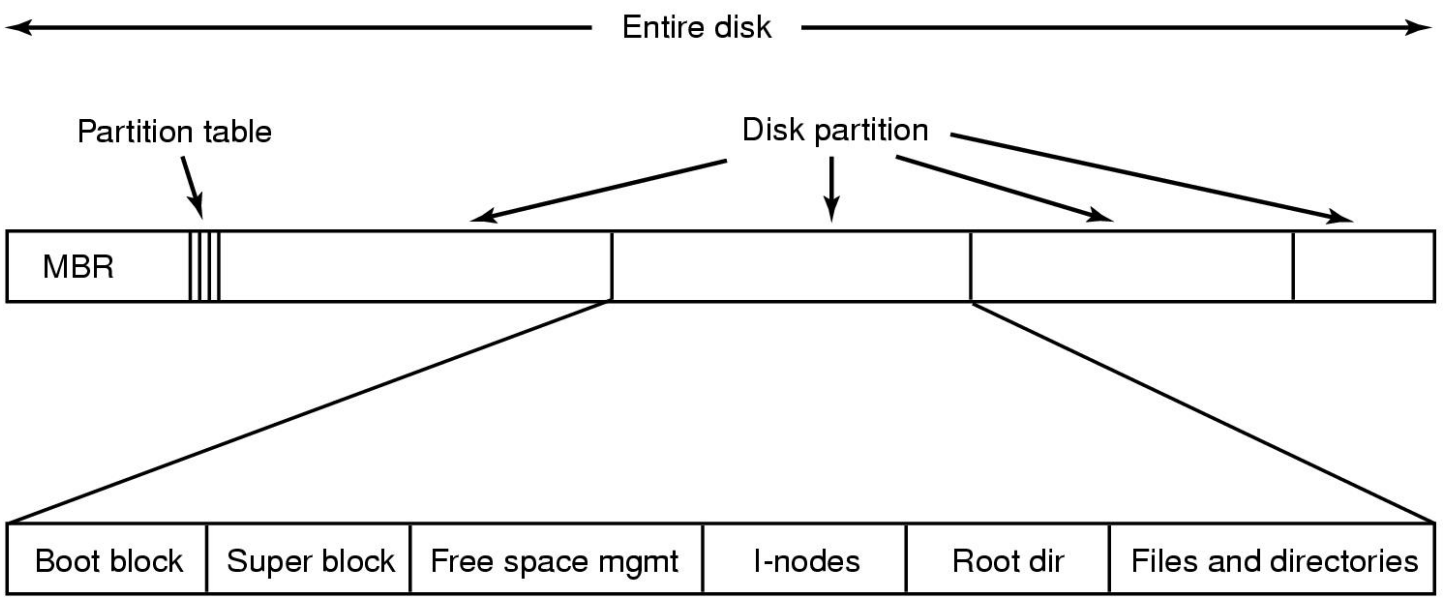
Directory Operations

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

File System Implementation

- Disks may contain one or more file systems
- Sector 0 is the Master Boot Record
  - Used to boot the computer
  - Contains a partition table with start and end address of each partition
- An active partition is chosen and the first sector (the *boot sector*) is read in memory and executed
- Each partition contains extra information used to manage the partition space
  - Free blocks, Used blocks
  - Root directories
  - ...

File System Implementation



Blocks

- Files and directories are allocated in *blocks*
- Block size is usually a multiple of the sector size
- Information about the blocks that compose a file must be managed
- File systems usually contain many small-size files and few big ones

Blocks

- Big block size (e.g., 32 KB)
  - Management does not require much space
  - Lots of wasted space
  - Performance improves (one read for each block)

- Small block size
  - Saves space
  - May requires large management data structure
  - Requires many reads

Implementing the File System

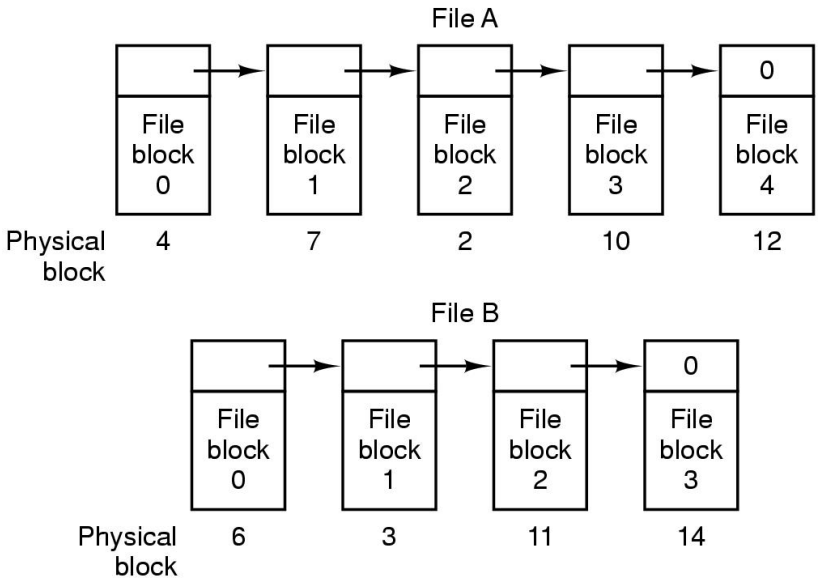
- Implementing files
  - Keep track of which blocks are used by a files
    - Contiguous space
    - Linked list
    - File Allocation Table (FAT)
    - I-nodes
- Implementing directories
  - Keep track of which files are in a directory
- Managing free lists
  - Manage free disk space

Contiguous Allocation

- Files are stored as a sequence of contiguous blocks
- File-to-blocks mapping is implemented with two numbers:
  - First block
  - Number of blocks used
- This schema supports sequential disk reads and delivers better performance
- Drawbacks:
  - Fragmentation
  - Changes in size require reallocation
- Used in CD-ROM (ISO 9660)

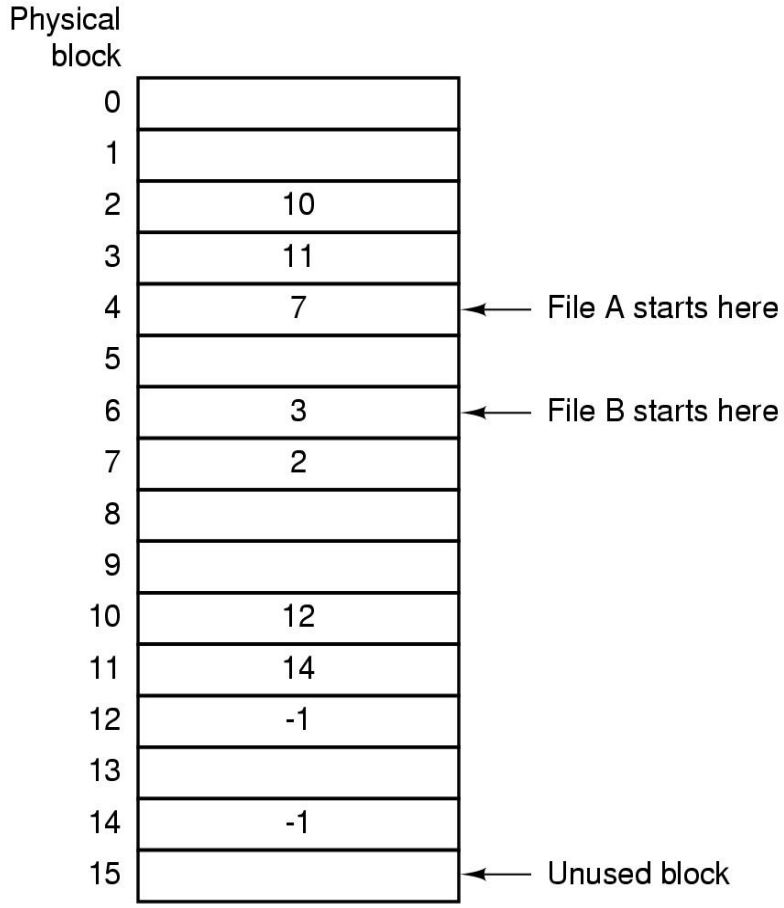
Linked Lists

- Each block contains a pointer to the next block in the file
- A file can be accessed by specifying the address of the first block and then following the list
- Drawback: Random access is expensive



File Allocation Table

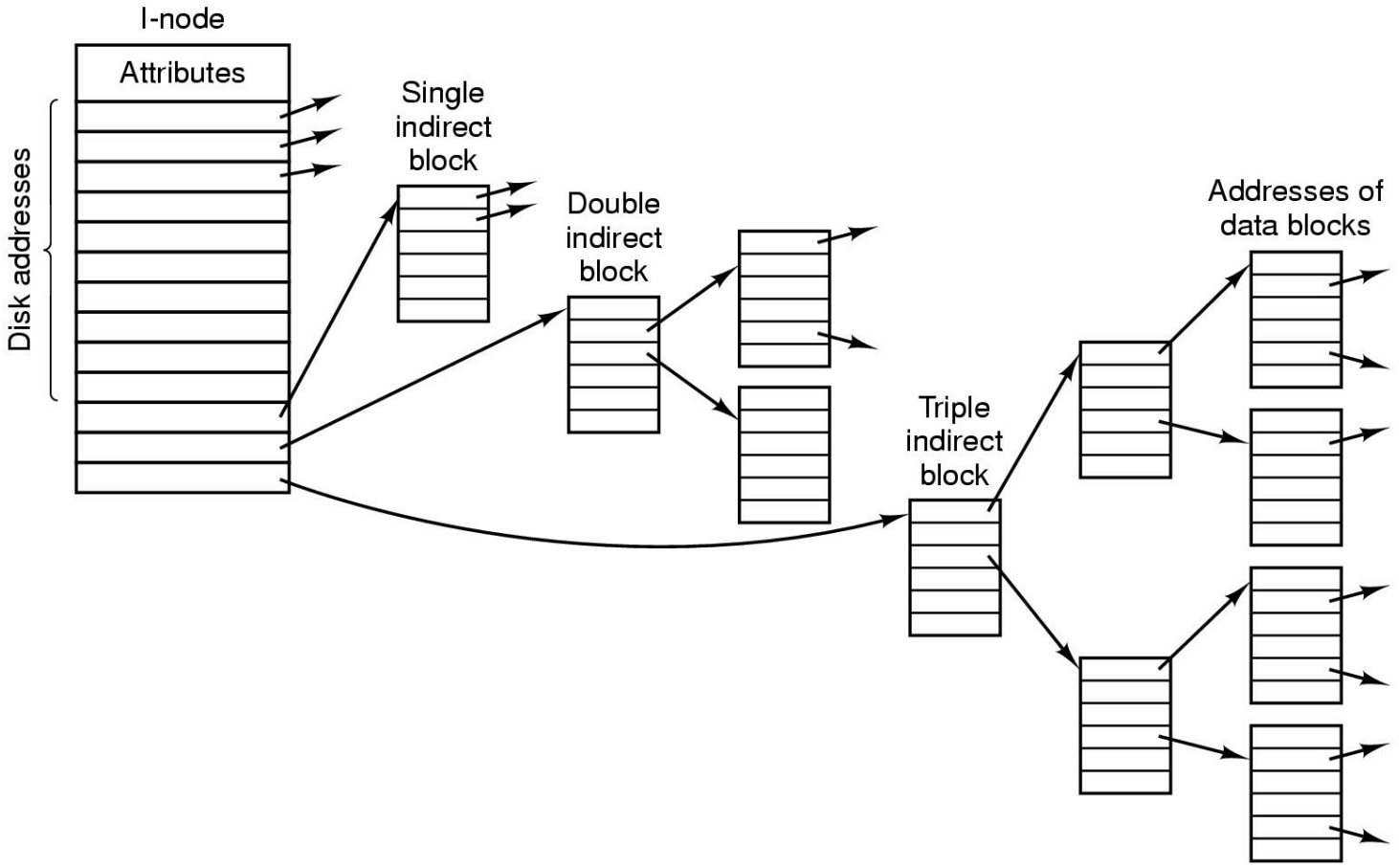
- Allocation table in RAM keeps track of “next block” information
- Random access is fast: Requires traversing the list, but in RAM
- A file can be accessed by specifying the address of the first block and then following the list
- Drawback:
  - Memory usage
    - Block size: 1KB
    - Partition size: 2 GB
    - Number of blocks: 2M
    - One FAT entry: 32 bits
    - FAT size 64MB!!



I-Nodes

- An i-node contains the file's attributes and a list of pointers to the blocks composing the files
- A first set of pointer represent direct addresses
- A second set of pointers are used to identify a block containing more block pointers (single indirection)
- A third set of pointers are used to implement double-indirection
- A fourth set of pointers are used for triple-indirection
- Advantage: I-node is in memory only when file is open

I-Nodes


I-node
Attributes
Disk addresses
Single indirect block
Double indirect block
Triple indirect block
Addresses of data blocks

Example

- Block size: 1KB
- Block address size: 4 bytes
- I-node contains 14 block pointers
  - 10 direct
  - 2 single indirect
  - 1 double indirect
  - 1 triple indirect
- File size will be
  - 4 KB, 100 KB, 500 KB, 64 MB, 1GB

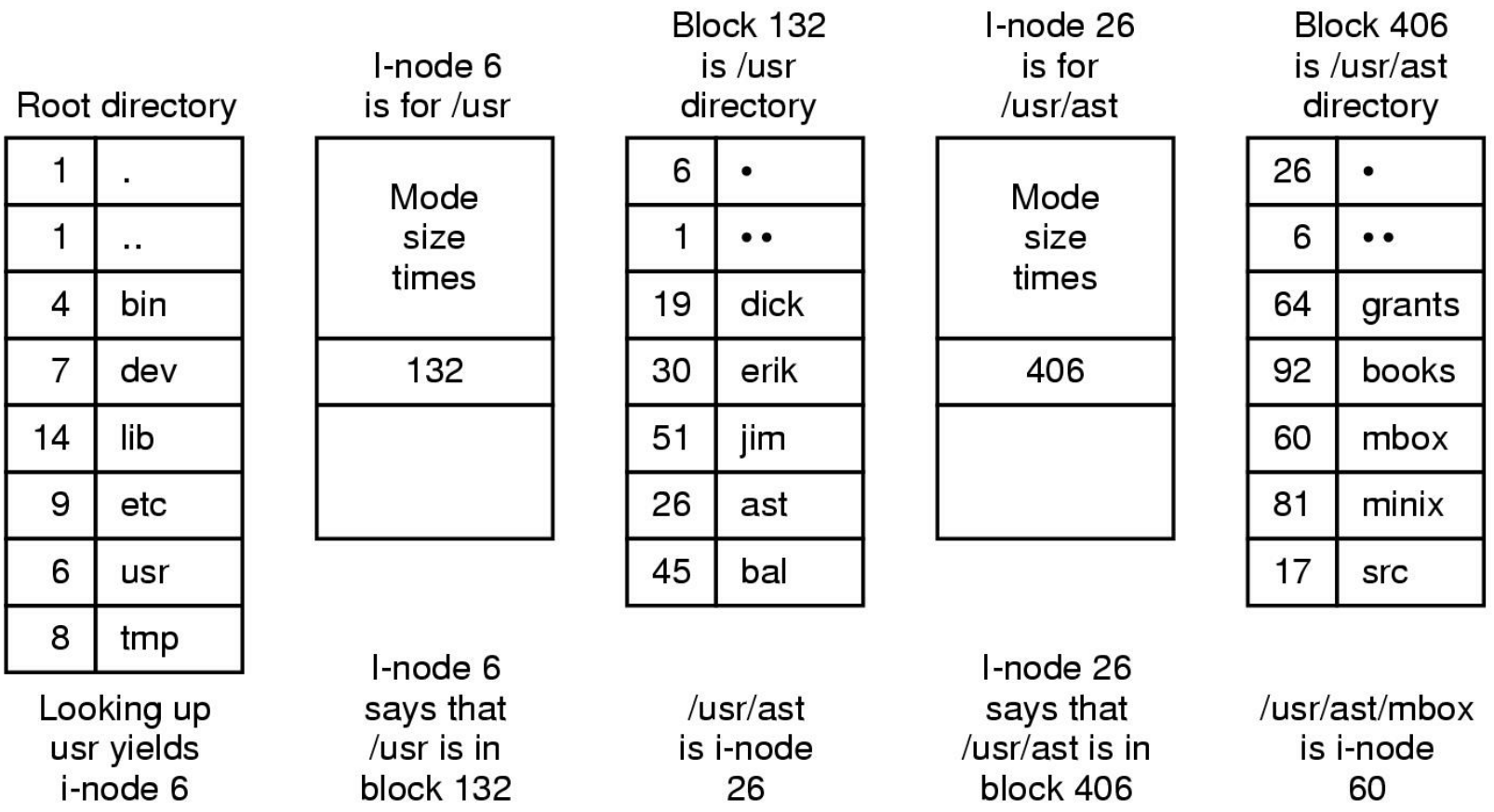
Solution

- 1 block can contain 256 block address
  - Single indirect: 256 KB
  - Double indirect: 256 * 256 = 65, 536 KB = 64MB
  - Triple indirect: 256 * 256 * 256 = 16,777,216 KB = 16 GB
- 10 KB: 10 direct
- 266 KB: 10 direct + 1 indirect
- 522 KB: 10 direct + 2 indirect
- 66,058 KB: 10 direct + 2 indirect + 1 d-indirect
- 16,843,274 KB: 10 direct + 2 indirect + 1 d-indirect + 1 t-indirect

Implementing Directories

- Directory provide a mapping between a symbolic name and the information used to retrieve the blocks composing the file
  - File name, First block
  - File name, I-node
- In some cases the directory entries are used to maintain the file's attributes

Accessing File /usr/ast/mbox

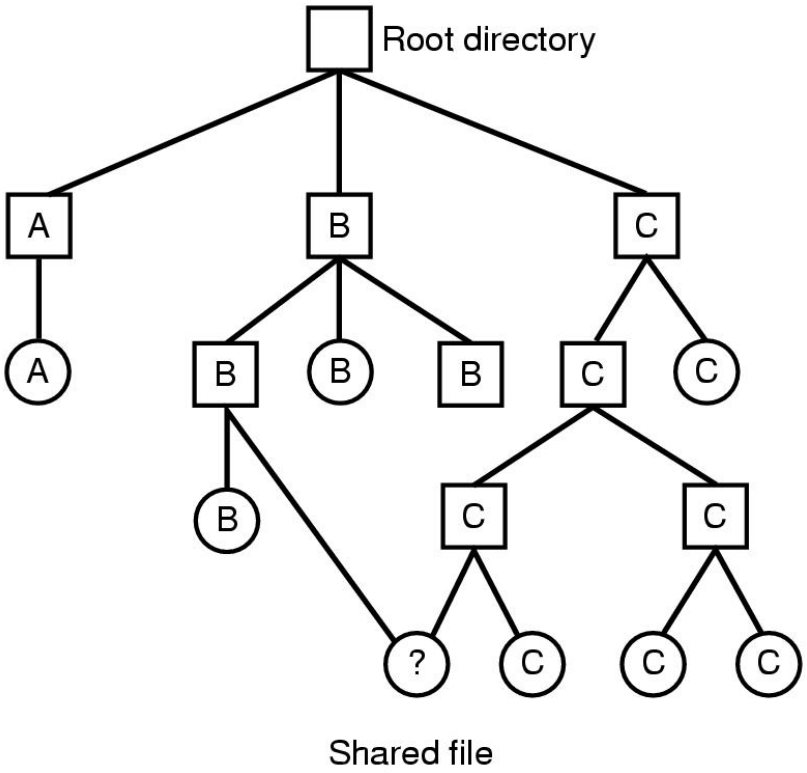


Shared Files

- In some file systems, file names referring to same file can be place in more than one directory

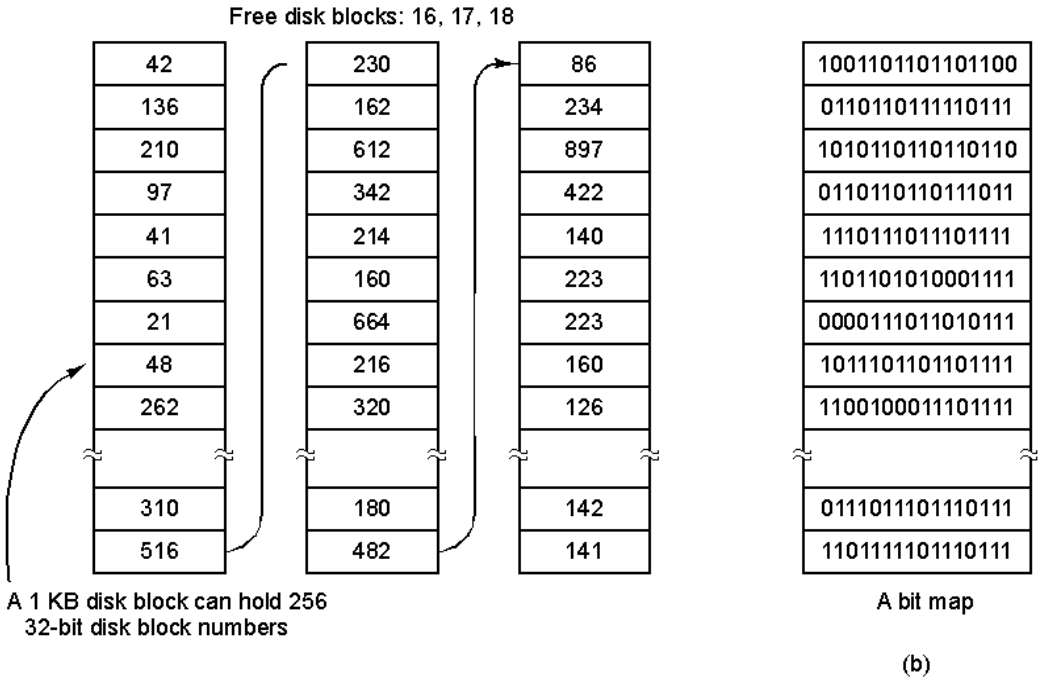
- Two types of references
  - Symbolic links
    - Entry contains actual path to be followed to access the file
    - Easy to manage
    - Generates small overhead
    - Can become invalid
  - Hard links
    - Entry directly points to file block information
    - No overhead
    - Deleting a file reference does not free the i-node
    - Must maintain counters of reference to file info (e.g., i-node)
    - must stay on same partition

Shared Files



Free Space Management

- Free blocks can be maintained using lists of blocks or bitmaps

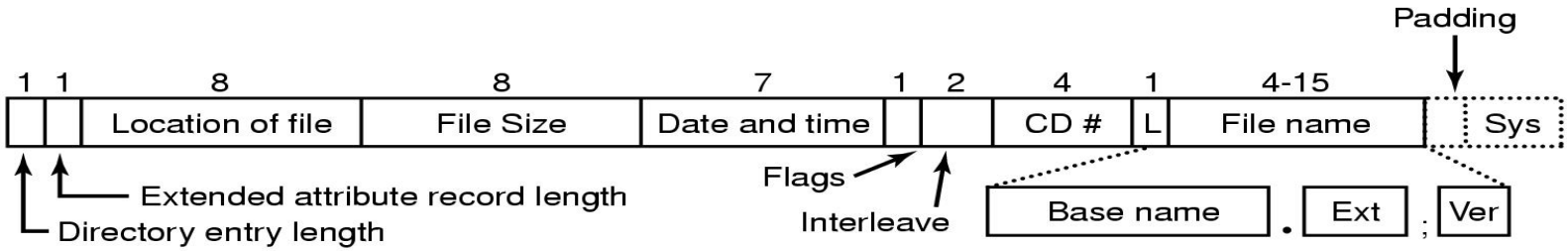


File System Performance

- File system performance is increased using
  - Block cache (e.g., managed in LRU mode)
  - Read-ahead
  - Minimizing disk arm motion

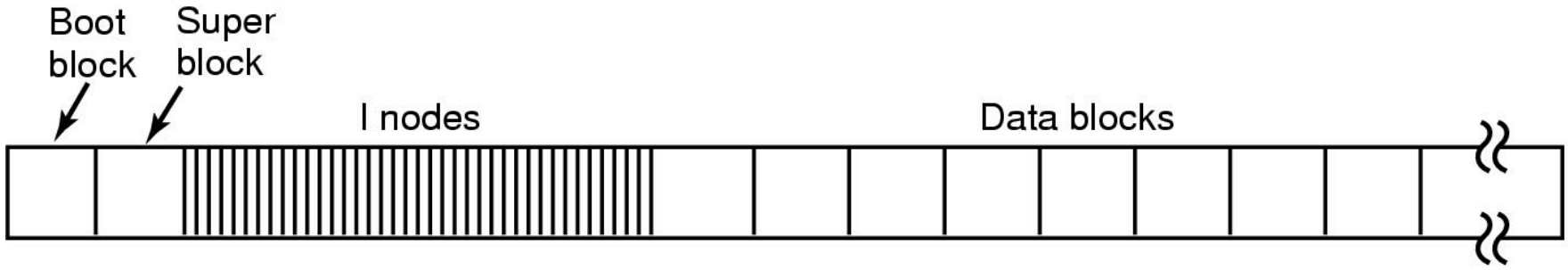
The ISO 9660 File System

- CD-ROM organized in sectors of 2352 bytes with a 2048 payload
- File system composed of a preamble (*primary volume descriptor*) containing authoring information and a pointer to the root directory
- Each directory is composed of a list of entries identifying contiguously allocated files



UNIX File System

- Disk layout in classical UNIX systems
  - Boot block contains bootstrap information
  - Superblock contains critical info such as
    - number of i-nodes
    - number of blocks



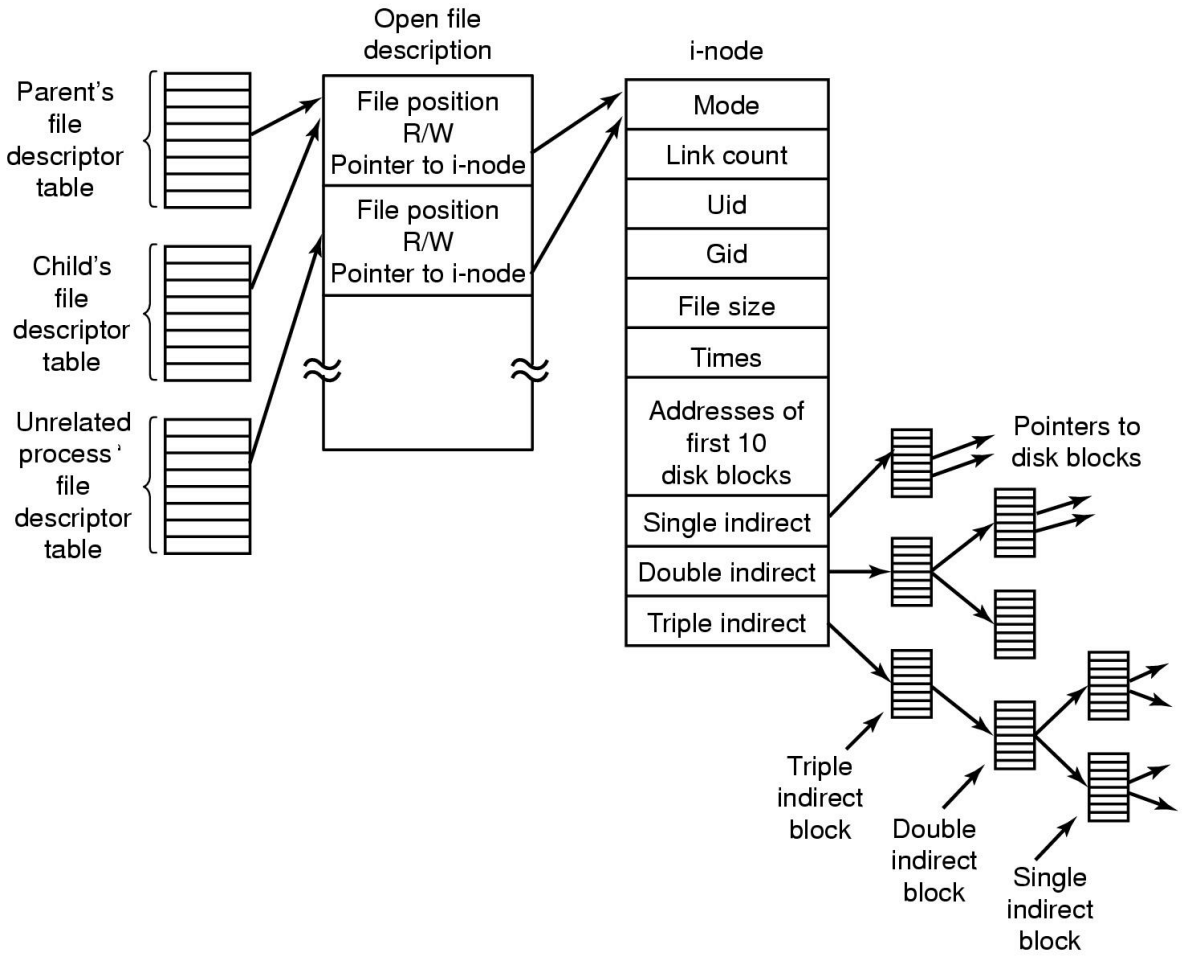
I-node in UNIX

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

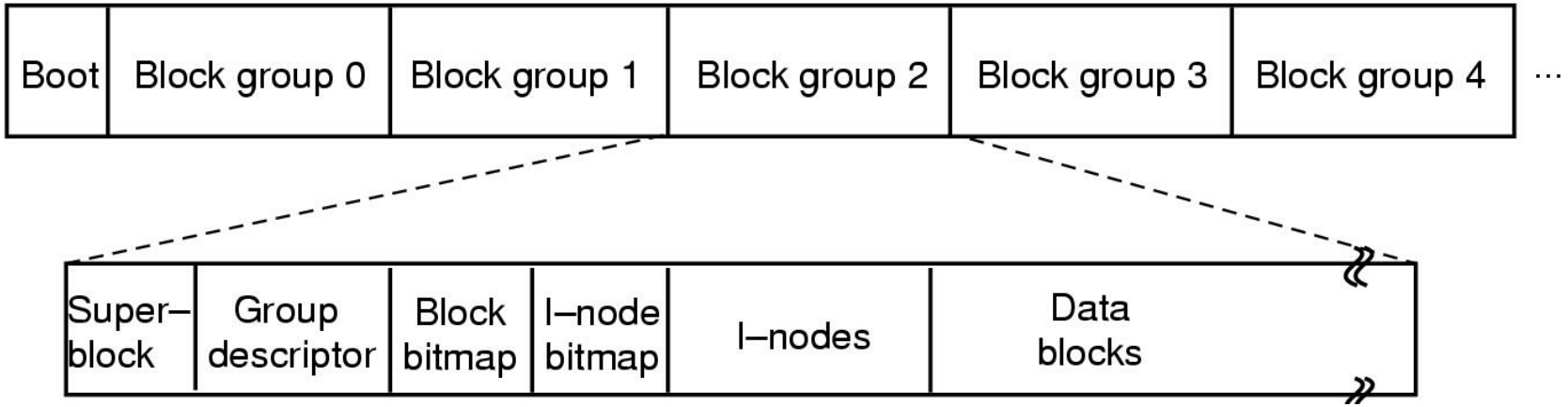
Accessing a File

- When a process requests the opening of a file, the i-node index is retrieved from the enclosing directory
- The actual i-node is retrieved from the i-node list on disk and put in the i-node table in the kernel
- An entry is created in the *open file description* table in the kernel
  - Contains current read/write position
  - Points to the i-node
- An entry is created in the *file descriptor table* in the process
  - Contains file descriptor that points to entry in kernel file description table
- This is done to allow parent/children to share file positioning

UNIX File System



The Ext2 File System



Mounting File Systems